

# CS422 - Programming Language Design

## Defining Concurrent Languages

Grigore Roşu

Department of Computer Science  
University of Illinois at Urbana-Champaign

We next show to design programming languages that allow concurrent execution threads, where threads can be created and terminated dynamically. Our approach is to enrich our previous continuation-based definitions of the **FUN** and **KOOL** languages with threads. As usual, we discuss our definitions in **K**, but these definitions can be automatically translated into **Maude**. By defining a concurrent language in **Maude**, we have two major benefits:

1. On the one hand, like before, we will get an *interpreter for free*. A *thread scheduler* for the defined programming language will be implicitly obtained as a consequence of Maude's internal rewriting rule application scheduler. In this version of our language definition we have no control on the thread scheduler.
2. On the other hand, and perhaps even more importantly, we get a *multithreaded program analysis tool* for free, which uses Maude's **search** command to find whether the program can reach certain good or bad states.

## Adding Syntax for the Concurrency Constructs

The syntax of the sequential parts of our multithreaded variants of **FUN** and **KOOL** remains unchanged. To both language definitions, we need to add syntax for the new concurrency-related features:

$$\text{spawn} : \text{Exp} \rightarrow \text{Exp}$$
$$\text{acquire} : \text{Exp} \rightarrow \text{Exp}$$
$$\text{release} : \text{Exp} \rightarrow \text{Exp}$$

The language construct **spawn** takes an expression and *creates a new thread* that will evaluate it. The new thread is also *passed the current environment* together with the expression to evaluate. The construct **spawn** returns immediately the value **unit**. Threads are therefore only used for their side effects. The new thread executes concurrently with the other thread(s), and they all have access to

the same store and can obviously *share data*.

There are multiple advantages of multi-threaded languages. One is that certain applications can be written more elegantly and abstractly. Another is speed of execution, when the hardware architecture provides multiple processors or threads. The major drawback of multi-threading is that multi-threaded programs are subject to an increased number of *unexpected non-trivial errors*.

Consider, for example, the following concurrent **FUN** program:

```
let x = 0
in {
    spawn(x := x + 1) ;
    spawn(x := x + 1) ;
    x
}
```

Two threads are spawn, each incrementing **x**. Therefore, three execution threads may live simultaneously before the **x** in the main

program is evaluated. What is the value returned by this program? It can be 0 when the main thread finishes first, 1 when only one of the additional threads finishes before the main program ends, or 2 when both threads end before the main program. Indeed, if we try the following search command in the new definition of `FUN` we will get the expected three solutions:

```
search eval(
  let x = 0
  in {
    spawn(x := x + 1) ;
    spawn(x := x + 1) ;
    x
  }
) =>* V:Value .
```

Threads that execute independently are of little or of no use at all. In order to be effective, threads need to *synchronize*. Our

synchronization approach is based on a simple *locking mechanism*.

To keep the definition of our concurrnet language extensions only interesting but not complex, we prefer to use available values as locks in the two language extensions discussed in this class. In **FUN**, we use ordinary integers as locks, while in **KOOL**, we allow any object to be a lock.

Two special language constructs, **acquire** and **release**, can be used by threads to acquire and to release locks. For example, an expression **acquire(12)** used in a thread **T** says that **T** tries to acquire lock **12**. At any given time, any lock can be held by *at most one thread*. Consequently, if lock **12** is already acquired by another thread then **T** waits until it is released.

The following is an example spawning two threads (so three execution threads in total). The two threads increment **x** in a synchronized manner to avoid data-races, and then mark their end

by assigning 1 to `a` and `b`, respectively. The main thread waits until both `a` and `b` are set and then outputs `x`. At the end of the execution of this program, `x` can only have the value 2:

```
let a = 0 and b = 0 and x = 0 in
{
  spawn(acquire lock(1); x := x + 1; release lock(1); a := 1);
  spawn(acquire lock(1); x := x + 1; release lock(1); b := 1);
  while (a eq 0) or (b eq 0) {};
  x
}
```

Note, however, that the program above *may not terminate*! Indeed, the main thread may take all the “CPU resources” while evaluating its `while` waiting loop, thus starving the other two threads that it spawned. Specialized schedulers are typically needed in a concurrent/multithreaded system to ensure that tasks/threads get a chance to eventually advance their computation.

The details of task/thread schedulers are typically not regarded as part of the design of a programming language (though are crucial for its effective implementation), so we do not discuss them in this class.

It is worthwhile mentioning that [Maude](#) provides some limited support for fair rewriting (see the command [frew](#)); however, this ensures fairness with respect to applications of rewrite rules, not necessarily with respect to “different” threads.



## K-based Definitions of Concurrent Languages

We next discuss the K-based semantics of our multithreaded languages.

- See pages 68–72 in the provided “K-report” for the definition of multithreaded [FUN](#);
- See pages 9–10 in the provided report on [KOOL](#) for the definition of multithreaded [KOOL](#).

## Examples using Multithreaded FUN

Let us next discuss several examples, which will clarify our intuitions with respect to the semantics of concurrent languages.

The following program can evaluate to any of the integers 0, 1 or 2, depending upon how long the write operations of the threads are delayed:

```
let x = 0
in {
    spawn(x := x + 1) ;
    spawn(x := x + 1) ;
    x
}
```

The above example also contains a data-race: it may be very well possible that the two threads read the value of `x`, then they both

increment it, and then they both write it. In that case, the final value of the `x` will be `1`. If one wants to see all the (three) values that the concurrent program above evaluates to then one just needs to give the search command:

```
search eval(
  let x = 0
  in {
    spawn(x := x + 1) ;
    spawn(x := x + 1) ;
    x
  }
) =>* V:Value .
```

Let us next consider the following program:

```
let a = 0 and x = 0
in {
    spawn(x := x + 1 ; a := 1) ;
    while (a eq 0) {} ;
    x
}
```

The above can either evaluate to `int(1)` or loop forever. It loops forever when the original thread takes all the CPU to evaluate its while loop, giving the created thread no chance to evaluate. The latter is said to *starve*. However, note that the program above still has a *finite state space*. Indeed, the while loop does not generate new states of the program, but just jumps from one state to another forever. Consequently, we can also use the search command on it to find all the values that it can evaluate to:

```

search eval(
  let a = 0 and x = 0
  in {
    spawn(x := x + 1 ; a := 1) ;
    while (a eq 0) {} ;
    x
  }
) =>* V:Value .

```

As expected, one will indeed find that there is only one value that the above can evaluate to, namely `int(1)`. If one wants to see all the states that this program can ever be in, including those which are not necessarily (final) values, then one can give the search command:

```

search eval(
  let a = 0 and x = 0
  in {
    spawn(x := x + 1 ; a := 1) ;
    while (a eq 0) {} ;
    x
  }
) =>* PV:[Value] .

```

In this case, one will see that there are six possible states that the program above can be in.

Let us next consider the following similar program, but where the main thread creates two child threads:

```
let a = 0 and b = 0 and x = 0
in {
    spawn(x := x + 1 ; a := 1) ;
    spawn(x := x + 1 ; b := 1) ;
    while (a eq 0) or (b eq 0) {} ;
    x
}
```

As before, this program may loop forever when the main thread starves the other two. However, when it terminates, it can return two possible values: `int(1)` or `int(2)`. That's because the two increments of `x` can lead to a data-race. An appropriate search command can reveal that. Also, a search for all the states will reveal that there are 59 states that this program can ever be in.

The following two programs, when terminate, evaluate to `int(3)`:

```
let a = 0 and b = 0 and c = 0
in {
    spawn(while(b eq 0){} ; c := b + 1) ;
    spawn(while(a eq 0){} ; b := a + 1) ;
    spawn(a := 1) ;
    while(c eq 0){} ; c
}
```

```
let a = 0 and b = 0 and c = 0
in {
    spawn(let rec l() = if not(b eq 0) then c := b + 1 else l()
          in l());
    spawn(let rec l() = if not(a eq 0) then b := a + 1 else l()
          in l()) ;
    spawn(a := 1) ;
    let rec l() = if not(c eq 0) then c else l() in l()
}
```



The following simple program can evaluate to any natural number:

```
let a = 0 and c = 0
in {
    spawn(a := 1) ;
    while(a eq 0) c := c + 1 ;
    c
}
```

That's because the main thread can wait in its while loop an arbitrary number of iterations incrementing therefore **c** an arbitrary number of times; if eventually the created thread sets **a** to **1** then the main thread exits its loop and outputs the current value of **c**. One can therefore use search to show that the above can evaluate to any specific natural number:

```
search [1] eval(  
  let a = 0 and c = 0  
  in {  
    spawn(a := 1) ;  
    while(a eq 0) c := c + 1 ;  
    c  
  }  
) =>* int(10) .
```

However, the state space of this simple program is now *infinite*!  
Therefore, it is necessary to use “**search [1]**” in the above command to instruct Maude to stop searching after it finds the first solution.

Let us next consider the following interesting program:

```
let c = 1
in {
    spawn(while (true) c := c + c) ;
    spawn(while (true) c := c + c) ;
    c
}
```

This was called the *thread game* by J Moore, and was claimed to potentially evaluate to any natural number. Can you prove it? The point here is that, in the case of concurrent programs,  $c + c$  is not equivalent  $2 * c$ ! Indeed, when evaluating  $c + c$ , one thread can read the first  $c$  and, right before it reads the second  $c$ , the other thread may write a new value to  $c$ . As usual, one may try to search that this program can evaluate to any specific natural number. Its state space is also infinite, so one cannot list the entire state space.

As already mentioned, threads and concurrency are of little use without synchronization. Let us next see how synchronization helps eliminate the undesired executions of concurrent programs.

The following program, when terminates, can evaluate to only one value now, `int(2)`:

```
let a = 0 and b = 0 and x = 0
in {
  spawn(acquire(1) ; x := x + 1 ; release(1) ; a := 1) ;
  spawn(acquire(1) ; x := x + 1 ; release(1) ; b := 1) ;
  while (a eq 0) or (b eq 0) {} ;
  x
}
```

That is because the data-race on `x` is not possible anymore.

As explained before, there are situations when the same thread can acquire the same lock several times during its computation. If that is the case, then one needs to ensure that the lock is also released by the thread as many times as it was acquired. The following is an example, part of a potentially larger one with many threads, in which one thread updating a shared variable with its factorial wants for obvious reasons exclusive access on the shared variable for the entire period of its computation. Note that the lock `l` is acquired as many times as the original value of `x`, and then released as many times as it was acquired:

```
let x = 5 and l = 1
in let rec f(y) = {
    acquire l ;
    if y eq 0 then {}
    else {
        x := x * y ;
        f(y - 1)
    } ;
    release l
}
in {
    f(x - 1) ;
    x
}
```

The following is the classical *dining philosophers* problem. Several philosophers dine at a round table. There is one fork in-between any two philosophers. To eat, a philosopher needs two forks.

The problem is to show that the philosophers may starve if they do not pick the forks wisely. Indeed, if all philosophers pick, for example, their left fork first and then wait for the second fork to be available, then they may wait forever for the second fork to be available. Indeed, the next search command will detect the deadlock. Note the use of `eval*`, stating that we are only interested in those states in which all the created threads terminated:

```

search eval*(
  let n = 3
  in let i = 0
      and f(x) = {
        acquire(x) ; acquire((x + 1) % n) ;
        --- eat
        release((x + 1) % n) ; release(x)
      }
      in {
        while not(i eq n) {
          spawn(f(i)) ;
          i := i + 1
        }
      }
) =>! PV:[Value] .

```



One (classical) way to fix the deadlock above is that the philosophers on odd positions pick their left fork first and then their right fork, while the philosophers on even positions pick their right fork first and then their left fork. This way, one can show that there is no way, that is unfortunate timing or delaying, for the philosophers to end up with just one fork waiting indefinitely for the other fork.

Indeed, the following fixed version of our program can be shown by the search command that it always terminates appropriately:

```

search eval*(
  let n = 3
  in let i = 0
      and f(x) = if x % 2 eq 1
                  then {
                      acquire(x) ; acquire((x + 1) % n) ;
                      --- eat
                      release((x + 1) % n) ; release(x)
                    }
                  else {
                      acquire((x + 1) % n) ; acquire(x) ;
                      --- eat
                      release(x) ; release((x + 1) % n)
                    }
      in {
          while not(i eq n) {
              spawn(f(i)) ;
              i := i + 1
          }
      }
) =>! PV:[Value] .

```